A General Framework for Complex Time-Driven Simulations on Hypercubes

David L. Meier, Kathleen L. Cloud,
Joan C. Horvath, Lynn D. Allan, Wayne H. Hammond
Jet Propulsion Laboratory

Heath A. Maxfield
California Institute of Technology

ABSTRACT

We describe a general framework for building and running complex time-driven simulations with several levels of concurrency. The framework has been implemented on the Caltech/JPL Mark IIIfp hypercube using the Centaur communications protocol. Our framework allows the programmer to break the hypercube up into one or more *subcubes* of arbitrary size (*task parallelism*). Each subcube runs a separate application using *data parallelism* and *synchronous* communications internal to the subcube. Communications between subcubes are performed with *asynchronous* messages. Subcubes can each define their own parameters and commands which drive their particular application. These are collected and organized by the Control Processor (CP) in order that the entire simulation can be driven from a single command-driven shell. This system allows several programmers to develop disjoint pieces of a large simulation in parallel and to then integrate them with little effort. Each programmer is, of course, also able to take advantage of the separate data and I/O processors on each hypercube node in order to overlap calculation and communication (*on-board parallelism*) as well as the pipelined floating point processor on each node (*pipelined processor parallelism*).

We show, as an example of the framework, a large space defense simulation. Functions (sensing, tracking, etc.) each comprise a subcube; functions are collected into defense platforms (satellites); and many platforms comprise the defense architecture. Software in the CP uses simple input to determine the node allocation to each function based on the desired defense architecture and number of platforms simulated in the hypercube. This allows many different architectures to be simulated. The set of simulated platforms, the results, and the messages between them are shown on color graphics displays. The methods used herein can be generalized to other simulations of a similar nature in a straightforward manner.

I. INTRODUCTION

Many applications in scientific computing cannot be solved with the homogeneous approach traditionally used with hypercube multicomputers. Solutions to inhomogeneous problems are required by such applications as electronic circuit simulations, war games, simulations of spacecraft systems, simulations of national or world economies, etc. Often such applications involve a degree of time-dependence. That is, the character of the solution evolves with time. We call such applications *inhomogeneous time-driven simulations* and they are characterized by the following features: 1) They are composed of *TASKS* with various degrees of workload. 2) Tasks communicate with one another to perform the simulation. 3) Each task has a *COMPUTATION CYCLE* which is repeated many times duration the simulation. 4) Each cycle has four phases: a) reception of data from other tasks, b) processing of that data, c) communication of results to other tasks, d) advancing simulation time τ. 5) Different tasks may take different amounts of *simulation time* to perform their computation cycles, as well as taking different amounts of *real time*.

We have developed a general simulation framework for building and running such inhomogeneous time-driven simulations on hypercubes. The goals of our framework are to:

1. run tasks in parallel for maximum speed-up;
2. load balance the processing power of the hypercube nodes so CPU-intensive tasks receive more CPU cycles than simple tasks;
3. keep tasks distinct so they can be added, deleted, or replaced at will -- even at run time (however, we do not support the addition, deletion, or migration of tasks *during* the simulation);
4. allow multiple instances of each task to be simulated, the number of such instances also being determined at run time;
5. develop a communication system which can
   a. determine which tasks communicate with each other and with what kind of data (at present we allow such dynamic configuration to occur only at run time, but we plan to support dynamic reconfiguration during the simulation in the near future),

0-8186-2113-3/90/0000/0117$01.00 © 1990 IEEE

b. react to the inclusion of additional task instances as well as the non-inclusion of other tasks by developing an appropriate communication graph (again supported only at the beginning of the run at present),

c. keep messages in proper simulation time and real time sequence, deliver them at the correct simulation time, and keep the system from deadlocking;

6. allow the simulation to be controlled by the user from a single location, despite its multifaceted character.

In this paper we describe the methods by which we have implemented such a simulation framework and then discuss, as an example, a large space defense simulation -- "Simulation 88" -- which makes use of at least five different levels of parallelism available in the JPL/Caltech Mark IIIfp hypercube. We believe that Simulation 88 is one of the most sophisticated applications run on a hypercube to date.

II. THE GENERAL HYPERCUBE SIMULATION FRAMEWORK

A. Mixed Task and Data Parallelism Using the Centaur Operating System

Goals 1 - 4 are achieved in the following manner. Each task is decomposed onto a *SUBCUBE* of the hypercube (*task parallelism*). As well as possible, the number of nodes in each subcube is kept approximately proportional to the task workload per computation cycle divided by the desired *simulation* time per cycle (the *throughput* of the subcube). (Of course, the number of nodes in each subcube must be a power of 2.) Each subcube has a designated master node, the *CORNER NODE*, which communicates with corner nodes of other subcubes.

Within each task the computation is generally homogeneous. Therefore, algorithm speed-up is accomplished using *data parallelism* algorithms, *i.e.*, the traditional homogeneous algorithms often proposed for hypercubes [1].

All communications, whether within or between subcubes, are handled by the *CENTAUR OPERATING SYSTEM*. [2] Within a subcube, the programmer uses fast synchronous communication subroutine calls (those from the so-called "crystalline operating system" or CrOS portion of Centaur). Between subcubes, specifically between corner nodes, and in communications with the outside world, the programmer uses asynchronous communication subroutine calls (those from the "Mercury" portion of Centaur). In Figure 1 we show a generic example of a 32-node hypercube decomposed into eight (8) separate subcubes, each of which is an instance of one of five distinct tasks.

Our scheme for decomposing the hypercube into subcube tasks is described as below. Consider the following input parameters:

D -- dimension of full hypercube

Δt_{1i} -- real time for task i to run one cycle on one hypercube node

$\Delta \tau_i$ -- desired simulation time for one cycle of task i

n_i -- number of instances desired for tasks i

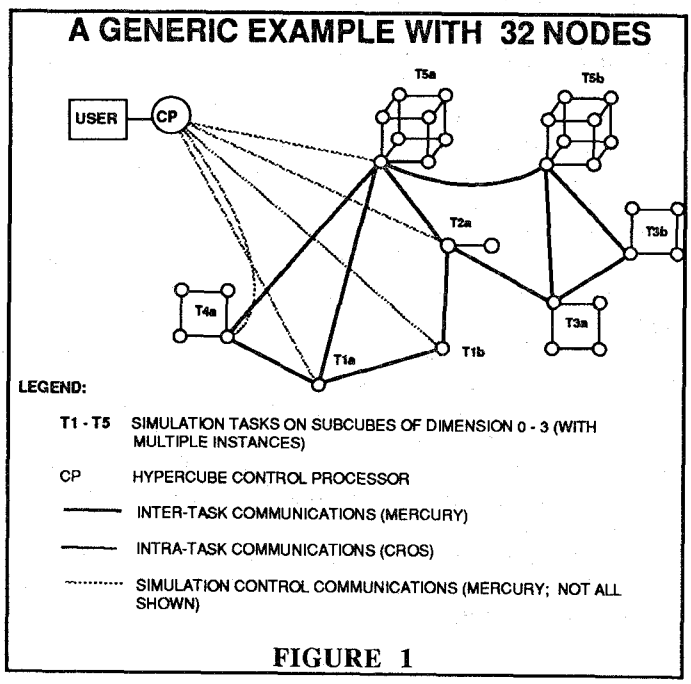

FIGURE 1

To solve the decomposition problem, one must solve for the dimension of each task's subcube d_i, subject to the following constraints:

Each task's throughput must be load balanced as well as possible: $TP_1 \cong TP_2 \cong TP_3 \cong \ldots$, where $TP_i = 2^{d_i} \Delta\tau_i / \Delta t_{1i}$

Each task has at least one node: $d_i \geq 0$

Tasks must fill hypercube: $\sum_i n_i 2^{d_i} = 2^D$

These constraints are satisfied by the following algorithm:

1. Initialize all $d_i = 0$
2. Compute the throughput TP_i of each subcube i

3. Choose the subcube **j** with the lowest throughput and compute the result of attempting to double the number of nodes for subcube **j**:

$$N_{test} = \sum_{i \neq j} n_i \, 2^{d_i} + n_j \, 2^{d_j + 1}$$

4. If $N_{test} \leq 2^D$, replace $d_j \leftarrow d_j + 1$; else freeze d_j, but continue searching for lowest throughput among other subcube tasks ($i \neq j$)

5. If all d_i have been frozen, exit the above loop, advancing to step 6; else to go step 2

6. If the final $N_{test} < 2^D$, then there are spare nodes left, but there is no task which needs them or which can be doubled to fill them. Instead, fill them with a null task.

B. Constructing the Communications Graph (Between Corner Nodes)

Once the hypercube has been partitioned into subcubes, the set of communication links among the subcubes -- the *communications graph* -- must be specified. This graph, and the communications calls made during the simulation, are the key elements of our simulation framework. They ensure that the correct data are passed between tasks at the proper simulation times so that the tasks can continue to perform their computations without deadlock.

One important requirement of the communication system is that it must be able to build a graph given the number of tasks and their instances available in the hypercube at run time. It would be very cumbersome for the user if he were required to manually reconfigure the communication links every time he added or deleted a single task instance, or changed a task's throughput, thereby changing the number of nodes devoted to it. We have therefore implemented a general scheme where the user specifies ***GENERAL COMMUNICATION LINKS***, which are valid under a wide variety of circumstances. These general links are then used by the framework to construct ***SPECIFIC COMMUNICATION LINKS*** at run time. The user need not know the number, size, or position in the hypercube of the subcube tasks in order to use this general scheme.

For each general link the user must specify:

The type of data being sent (a master list of allowed message types must be defined and be made part of the framework);

The sending task type (but not the instance nor the node number);

The receiving task type;

A ***COMMUNICATION RULE*** specifying to which of the several possible instances of sending tasks the receiving task should ***LISTEN*** for this message type.

Note that this "simulation mapping" process requires algorithms specific to the simulation being performed. It must be modified for each new simulation being developed.

The specification of "to whom to listen", rather than "to whom to send", is important. One can be derived from the other, but it is much easier to construct the latter from the former and insure that all tasks receive the data they need. In addition, by avoiding multiple sources of data for each type, this method insures that most (if not all) messages sent will be picked up and used by the receiving subcube. (This is useful as hypercube nodes have a finite amount of memory and cannot afford to leave a large number of unread messages.) There is also no need to create data arbitration algorithms for each communication reception to handle the case when more than one message of a given type arrives. However, this feature limits our framework to those simulations where tasks have only one source for any given data type. (Of course, additional data types can be defined to maintain the flexibility needed in most situations.)

At run time the general links are used to construct the specific communication links. A specific link is defined by 1) the sending subcube's corner node number and task type, 2) the receiving subcube's corner node number and task type, and 3) the message type. All links involving a single corner node are stored on that node in a lookup table. When a SEND of a certain message type is executed by a task, the intended receiving subcubes' type must also be specified in the communication call. (Only one call is needed to send to all subcubes of the same task type, but multiple sends must be performed if the same message type is intended for more than one task type.) The framework software then looks in the table for all links with the proper receiving task type and delivers the message to them. If no links satisfy the criteria, the call is ignored, but an error code is returned. Likewise, when a RECEIVE of a message type is executed, the software first checks the lookup table to see if the link has been defined.

The above scheme avoids deadlock in two cases: when a specific link is defined, but SEND and/or RECEIVE are not called; and when a link is not defined, and SEND or RECEIVE is called. Nevertheless, the tasks must be coded carefully as problems can still occur. Deadlock will occur if a link is defined and a RECEIVE is called by one task, but the sending task specified by the link has not called a SEND. Data overflow can occur if a link is defined and a SEND is called by one task, but the corresponding RECEIVE is not. The message queue on the receiving task then grows linearly with time.

C. The Synchronization of Tasks Using Message Passing, The Control of Simulation Time, And The Assurance of Task Parallelism

A message-passing system works properly only if the messages are sent and received at the proper times. Therefore, message passing cannot be considered without also considering the flow of time in the simulation and the method by which the tasks are synchronized. In our framework, synchronization is controlled by the message passing, just as it is in homogeneous applications, by forcing the receiving task to wait until it has received a message which satisfies certain criteria.

Each task has its own internal clock which advances simulation time in fixed increments of $\Delta\tau_i$. (Simulation time increments of different task types do not have to be the same.) Furthermore, in addition to the normal message header information which Centaur places on the message, our framework also time tags each message with the simulation time at which it is sent. A message sent by task j at simulation time τ_j and received by task i at time τ_i is accepted only if its time tag τ_j is in the interval

$$\tau_i - \alpha\,\Delta\tau_j \le \tau_j < \tau_i + (1-\alpha)\,\Delta\tau_j \ .$$

(α is a parameter which describes the type of message acceptance: $\alpha=1$ denotes backward-biased, $\alpha=0$ denotes forward-biased, and $\alpha=1/2$ denotes time-centered acceptance.) Messages which are accepted are read from the queue but not discarded; only messages with tags $\tau_j < \tau_i - \alpha\Delta\tau_j$ are deleted. Note that the acceptance time interval is determined by the sender's simulation clock "tick" ($\Delta\tau_j$) and not the receiver's. This avoids deadlock regardless of whether the ratio $\Delta\tau_j / \Delta\tau_i$ is less than or greater than unity. If a message of the correct type, sending node, and time tag is not in the queue, the receiving task waits until one arrives. This synchronizes the tasks.

It is possible with such a synchronization scheme to force the tasks to execute in a sequential fashion and not in parallel! That is, it is possible that only one task at a time is performing any computations and that all the other tasks are waiting, especially if the communication graph has one or more closed loops embedded in it. This serial processing can be avoided if each task executes its operations in each cycle in a particular order:

1. Send all messages; if there is no data to send, still send a null message (header);
2. Receive all messages from other tasks;
3. Perform CPU-intensive work
4. Advance simulation time by $\Delta\tau_i$ seconds

Sending all messages first "primes the pump" and allows other tasks to continue executing in parallel, especially when closed loops exist in the graph. Advancing the simulation time after the computation emulates the passing of simulation time during the computation portion of the cycle.

D. Centralized Simulation Control (The C3PO System)

Control of the execution of the simulation is provided by a program running in the Control Processor of the hypercube (see Figure 1): *C3PO* (Command and Parameter Processor for Program Organization). After the hypercube is partitioned, and before the communication links are set up, each subcube task defines a set of parameters and commands which control that task. (Typical parameters are names of initialization files, printing and plotting flags, etc.; typical commands are initialization, starting the execution of the cycles in each task, and commands which alter the simulation during execution such as shutdown.) The parameters and command names and types are sent up to C3PO in the CP where they are stored in a symbol table. All tasks then listen to C3PO for commands and continue to do so even while executing other commands.

A one-word command issued by the user at the C3PO prompt will execute a subroutine on all subcubes which recognize that command. In addition to commands, the C3PO interpreter also executes a C-like language. Parameter values may be altered at the C3PO level with assignment statements, C3PO functions, etc. Each command issued will use the latest values of the parameters.

For sophisticated simulations the C3PO program itself can be a task with its own set of parameters and commands. This is most useful during the pre-initialization phase of a simulation when the partitioning of the hypercube into tasks is determined.

III. APPLICATION TO A COMPLEX STRATEGIC DEFENSE SIMULATION: SIMULATION 88

We have used the above framework to construct a detailed simulation of a strategic defense system. This simulation, called Simulation 88, is an emulation of a portion of a constellation of missile sensors, trackers, battle managers, and weapons platforms. Simulation 88 is composed of the following major tasks, each of which is a separate C program: SWIR (short-wave infrared) sensor; tracker of SWIR sensor data capable of stereo processing; LWIR (long-wave infrared) sensor; tracker of LWIR sensor data capable of stereo processing; a global engagement manager which allocates weapons in the arsenal based on ability to engage and the probability of kill; a fire control module which schedules weapon release and performs guidance; an environment generator which launches the threat, flies the

SDI platforms, and generally takes care of functions performed by the enemy or by nature; and a simulation monitor which doubles as the null task when not running on node 0 of the hypercube. In addition to being able to communicate with one another to assess and respond to the threat, most tasks can also open one or more windows on external color graphics workstations for viewing the simulation progress. The amount of C code running on these workstations is nearly equal to that running in the hypercube.

Simulation 88 has been designed and implemented in an unclassified environment. However, it is parameterized (through the use of C3PO parameters and initialization files) so that it can be run in a classified manner in the proper environment.

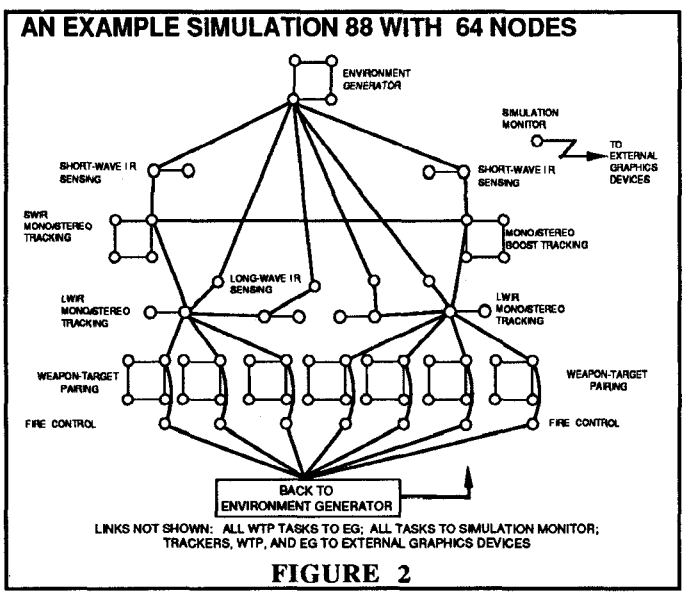

FIGURE 2

A run of Simulation 88 is uniquely determined by a configuration file which defines 1) which tasks are active, 2) how tasks are bundled together to form SDI platforms, 3) the total number of platforms of each type and their orbits, 4) how tasks communicate (the general links), 5) how many platforms of each type we wish to emulate in the hypercube (the rest are simulated in lower fidelity), and 6) how large a hypercube we wish to use for the simulation. All this information is parsed by the C3PO program before the hypercube is booted. After all executable code is downloaded into the hypercube, the specific platforms to be emulated are chosen from the constellations according to which ones can fight the battle best. All communication links between platforms are constructed, but only that subset which involves the chosen platforms is used for actual Centaur communications between tasks. Figure 2 shows the node allocation which results from a typical 64-node run. Increasing the cube dimension to seven (7), for example, would not change the number of modules but would change the numbers of nodes allocated to each.

Simulation 88 makes use of at least five different levels of parallelism:

Multi-machine parallelism: graphics processing and display occur in parallel with the hypercube simulation computations;

Task parallelism within the hypercube: the simulation is subdivided into subcubes; C3PO in the CP is also a task;

Data parallelism *within each subcube of dimension* $d_i > 0$: each task occupies 2^{d_i} nodes;

Intra-node parallelism: each task's code runs in the 68020/68882 processor or in the Weitek floating point processor; the Centaur communications is performed in parallel by a separate 68020 on each hypercube node;

Pipelined parallelism: some tasks execute their code in the Weitek floating point processor of the Mark IIIfp hypercube; this processor accomplishes parallelism on a machine instruction level.

IV. ACKNOWLEDGEMENTS

The work described in this paper was carried out at the Jet Propulsion Laboratory, under contract with the National Aeronautics and Space Administration.

Simulation 88 itself is a much larger project than just the results reported herein. In addition to the above authors, the following contributed substantially to the individual pieces of software run on the various subcubes: environment generation -- R. Yeung; sensing / tracking -- T. Gottschalk, R. Yeung; weapon-target pairing / fire control -- D. Payne, E. Leaver, J. Steinman. The color graphics screens were developed by J. Lathrop, R. Iwashina, M. Pomerantz, and L. van Warren. We are also grateful for the considerable assistance given by members of the Hypercube system software team (R. Lee, C. Goodhart, L. Craymer, J. Crichton, N. Meshkaty, and B. Zimmerman) and of the hardware team (J. Peterson, M. Pniel, and D. Smith). The JPL Hypercube project is managed by D. Curkendall, the teams are managed by J. Fanselow, and the applications team is managed by D. Rogstad.

V. REFERENCES

[1] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D., *Solving Problems on Concurrent Processors*, Prentice Hall, Englewood Cliffs, N.J., 1988.

[2] Goodhart, C. and Lee, R. "Centaur: A Mixed Synchronous /Asynchronous Communication Protocol for the Mark III Hypercube", Fourth Conference on Hypercubes, Concurrent Computers, and Applications, Monterey CA, Mar 6-8, 1989.